UCRL-ID-137373

AMR++: Object-Oriented Parallel Adaptive Mesh Refinement

B. Philip, D. Quinlan

February 2, 2000

U.S. Department of Energy
Lawrence Livermore National Laboratory

DISCLAIMER

Work performed under the auspices of the U. S. Department of Energy by the University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

AMR++: Object-Oriented Parallel Adaptive Mesh Refinement

Bobby Philip and Dan Quinlan

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, Livermore, CA, USA, 94550
{bobbyp,dquinlan}@llnl.gov

February 2, 2000

Abstract

Adaptive mesh refinement (AMR) computations are complicated by their dynamic nature. The development of solvers for realistic applications is complicated by both the complexity of the AMR and the geometry of realistic problem domains. The additional complexity of distributed memory parallelism within such AMR applications most commonly exceeds the level of complexity that can be reasonable maintained with traditional approaches toward software development.

This paper will present the details of our object-oriented work on the simplification of the use of adaptive mesh refinement on applications with complex geometries for both serial and distributed memory parallel computation. We will present an independent set of object-oriented abstractions (C++ libraries) well suited to the development of such seemingly intractable scientific computations. As an example of the use of this object-oriented approach we will present recent results of an application modeling fluid flow in the eye. Within this example, the geometry is too complicated for a single curvilinear coordinate grid and so a set of overlapping curvilinear coordinate grids[1] are used. Adaptive mesh refinement and the required grid generation work to support the refinement process is coupled together in the solution of essentially elliptic equations within this domain. This paper will focus on the management of complexity within development of the AMR++ library which forms a part of the Overture object-oriented framework[2] for the solution of partial differential equations within scientific computing.

1 Introduction

Adaptive mesh refinement is a numerical technique for locally tailoring the resolution of computational grids. AMR permits the addition of finer grids to the global computational grid in an adaptive way so as to permit locally more accurate computations or the removal of global error introduced by local singularities. Through the introduction of mesh points in a way specific to the computation typically one to two orders of magnitude better efficiency can be obtained. Thus AMR approaches are extremely attractive but offset by largely overwhelming complexity in their implementation.

Fortunately, AMR as a numerical technique is largely independent of the equations being solved, though numerous numerical and algorithmic issues are involved and are the subject of significant research. The AMR infrastructure, if reasonablely designed, can be similarly made independent of the equations being solved.

Unfortunately, AMR is not common place due largely to its inherent complexity (even in a serial environment!). While numerical issues concerning the best ways to handle refinement interfaces are extremely important, the development of parallel AMR has not become common place because its software

[1] [?]verturePaper
[2] `www.llnl.gov/casc/Overture`

complexity has far out striped improvements in the software productivity required for its implementation and use. A special role for libraries specific to AMR seems obvious, so that the complexity of the AMR process can be properly encapsulated[3]. The development of parallel AMR is of course even more problematic and tedious because of the dynamic distribution of data, and non-regular communication patterns that adaptive solvers necessitate. The development of elliptic solvers for adaptively refined grids is still more complex because of the global nature of the elliptic solution process in the more efficient elliptic methods (e.g. multigrid solvers). AMR represents an impressive opportunity to demonstrate a potentially significant simplifying object-oriented mechanism to build adaptive mesh refinement applications. In this paper we present the work we are doing to couple multiple complexities: adaptive mesh refinement, parallel computation, and complex geometric problem domains. The approach is object-oriented, but this does not make it novel.

The uniqueness of our approach is in how we have developed a hierarchy of abstractions within the Overture framework that work together to build many different sorts of applications and permit adaptive mesh refinement to be built additionally. This work is work we make available in AMR++ as part of the publicly available Overture framework.

1.1 Goals and Restrictions

AMR++ is designed to represent an object-oriented implementation of the infrastructure for adaptive mesh refinement. Details of specific solution algorithms are intended to be placed at higher levels (within higher level abstractions). Thus, AMR++ is not tailored to be specific to any specific class of PDEs. Future work may focus on abstractions specific to different types of equations: time-dependent and steady-state. Such work would further simplify the existing AMR applications, but in an essentially different way.

1.2 Related Research

Adaptive Mesh Refinement research is being done by several other groups as well, including Lawrence Berkeley Laboratory, SAMRAI at Livermore National Laboratory, HDDA/DAGH at University of Texas. In most cases the work thus far excludes handling complex geometries, though some are working on internal boundary mechanisms to address geometric requirements. The original work on AMR was done by Marsha Berger in the early 1980's. Other projects have also addressed adaptive mesh refinement, but none combined AMR with parallelism and support for complex geometry.

2 Principle Object-Oriented Abstractions

AMR++ leverages the objects developed within other parts of the Overture framework, this include the main class categories that make up **Overture** :

- **Arrays** describe multidimensional arrays using A++/P++. A++ provides the serial array objects, and P++ provides the distribution and interpretation of communication required for their data parallel execution.
- **Mappings** define transformations such as curves, surfaces, areas, and volumes. These are used to represent the geometry of the computational domain.
- **Grids** define a discrete representation of a mapping or mappings. These include single grids, and collections of grids; in particular composite overlapping grids. The adaptive mesh refinement on curvilinear coordinate grids is possible because these grids store there mapping internally, thus making it available during the numerical solution when adaptive mesh refinement is done.
- **Grid functions** storage of solution values, such as density, velocity, pressure, defined at each point on the grid(s). Grid functions are derived from A++/P++ array objects.
- **Operators** provide discrete representations of differential operators and boundary conditions
- **Grid generation** [?]: the Ogen overlapping grid generator automatically constructs an overlapping grid given the component grids.

- **Plotting** a high-level interface based on OpenGL allows for plotting **Overture** objects.

2.1 Array Operations

A++ and P++ are array class libraries for performing array operations in C++ in serial and parallel environments, respectively.

A++ is a *serial* array class library similar to FORTRAN 90 in syntax, but not requiring any modification to the C++ compiler or language. A++ provides an object-oriented array abstraction specifically well suited to large-scale numerical computation. It provides efficient use of multidimensional array objects which serves to both simplify the development of numerical software and provide a basis for the development of parallel array abstractions. P++ is the *parallel* array class library and shares an identical interface to A++, effectively allowing A++ serial applications to be recompiled using P++ and thus run in parallel. This provides a simple and elegant mechanism that allows serial code to be reused in the parallel environment.

P++ provides a data parallel implementation of the array syntax represented by the A++ array class library. To this extent it shares a lot of commonality with FORTRAN 90 array syntax and the HPF programming model. However, in contrast to HPF, P++ provides a more general mechanism for the distribution of arrays and greater control as required for the multiple grid applications represented by both the overlapping grid model and the adaptive mesh refinement (AMR) model. Additionally, current work is addressing the addition of task parallelism as required for parallel adaptive mesh refinement.

Here is a simple example code segment that solves Poisson's equation in either a serial or parallel environment using the A++/P++ classes. Notice how the Jacobi iteration for the entire array can be written in one statement.

```
  // Solve u_xx + u_yy = f by a Jacobi Iteration
Range R(0,n)                    // define a range of indices: 0,1,2,...,n
floatArray u(R,R), f(R,R)       // declare two two-dimensional arrays
f = 1.; u = 0.; h = 1./n;       // initialize arrays and parameters
Range I(1,n-1), J(1,n-1);       // define ranges for the interior

for( int iteration=0; iteration<100; iteration++ )
  u(I,J) = .25*(u(I+1,J)+u(I-1,J)+u(I,J+1)+u(I,J-1)-f(I,J)*(h*h)); // data parallel
```

The use of the array abstraction provides a particularly simple way to encapsulate data parallelism. The execution is not strictly data parallel but is SPMD (simulating data parallel execution), this provides a richer context in which we can mix data parallelism and task parallelism??.

2.2 Grid Generation

Overture has support for the creation of overlapping grids for complicated geometries. The process of generating an overlapping grid consists of two basic steps. In the first step a number of component grids are generated. Each component grid represents a portion of the geometry. The component grids must overlap but otherwise can be created locally. **Overture** provides a collection of `Mapping` classes that can be used to generate component grids including splines, NURBS, bodies of revolution, hyperbolic grid generation, elliptic grid generation, trans-finite interpolation and so on. In addition we are working on methods for reading files generated by CAD programs and generating grids.

Given the component grids, the overlapping grid then is constructed using the `Ogen` grid generator. This latter step consists of determining how the different component grids interpolate from each other, and in removing grid points from holes in the domain, and removing unnecessary grid points in regions of excess overlap. `Ogen` requires a minimal amount of user input.

As will be explained in more detail else where in the paper, the use of adaptive mesh refinement on the component grids (curvilinear coordinate grids) requires grid generation. Since the grid generator is an integral part of the Overture framework the grid generation can be called as required within the numerical solution phase of the computation where the adaptive refinement takes place.

3 Writing PDE solvers

This example demonstrates the power of the **Overture** framework by showing a basically complete code that solves the partial differential equation (PDE)

$$u_t + au_x + bu_y = \nu(u_{xx} + u_{yy})$$

on an overlapping grid.

The `PlotStuff` object is used to interactively plot contours of the solution at each time step .

```
int main()
\{
  CompositeGrid cg;                          // create a composite grid
  getFromADataBaseFile(cg,"myGrid.hdf");     // read the grid in
  floatCompositeGridFunction u(cg);          // create a grid function
  u=1.;                                      // assign initial conditions
  CompositeGridOperators op(cg);             // create operators
  u.setOperators(cg);
  PlotStuff ps;                              // make an object for plotting
  // --- solve a PDE ----
  float t=0, dt=.005, a=1., b=1., nu=.1;
  for( int step=0; step<100; step++ )
  \{
    u+=dt*( -a*u.x()-b*u.y()+nu*(u.xx()+u.yy()) );
    t+=dt;
    u.interpolate();                 // interpolate overlapping boundaries
    // apply the BC u=0 on all boundaries
    u.applyBoundaryCondition(0,dirichlet,allBoundaries,0.);
    u.finishBoundaryConditions();
    ps.contour(u);                    // plot contours of the solution
  \}
  return 0;
\}
```

The example solves the time-dependent equation explicitly. Other class libraries within the **Overture** framework simplify the solution of elliptic and parabolic equations, the linear systems generated can be solved using any of numerous numerical methods as appropriate including multigrid, and methods made available within a number of external dense and sparse linear algebra packages including PETSc, and others. These are wrapped into the elliptic solver library (Oges) within **Overture** .

These abstractions serve to simplify the development of scientific applications because each one addresses an orthogonal complexity. The representation of an application in terms of such abstractions is flexible because none of the abstractions overlap in capabilities with one another (in this sense they are orthogonal) and because together they allow the complete application to be expressed (the mathematical concept similar to completion). To be clear, the abstractions presented are insufficient for all scientific applications, they handle our own applications and additional abstractions would likely be required to address a larger set of scientific applications.

4 What Makes AMR Difficult

In this section we present why AMR is a difficult problem and worthy of a well planned object-oriented attack. In each case we explain briefly how we address this complexity. The use of the previously describe abstractions is key to addressing these complexities within the AMR++, specifically the use of these abstractions provides substantial code reuse in the development of AMR++. The use of AMR++ and lover level abstractions within the rest of Overture provides applications with additional and substantial code reuse again.

4.1 Inappropriate mixture of different levels of abstractions

Fundamentally, a parallel AMR application must represent many levels of abstraction are mixed together within parallel AMR. This adds to the complexity of the process. Many features must be present within any sufficiently general parallel AMR framework (library):

- data-distribution mechanisms
- dynamic redistribution mechanisms
- support for parallel communications (e.g. explicit MPI calls)
- support for load balancing
- support for error estimation
- support for regridding (manual or automated)
- performance
- Grid generation
- require for initial problem setup
- required for addition of local refinement
- moving grid support
- support for finite-difference and finite volume discretizations
- Customizable regridding
- Simple Interfaces

The requirements of AMR for general applications greatly mix both low level details (e.g. parallelism, performance requirements, etc.) with higher level issues (e.g. grid generation, load-balancing, regridding, etc.). The requirement to fit these into a single application greatly strain the ability to support and maintain such applications long term. That scientific applications are regularly built that mix low-level parallel communication with higher-level numerical methods does not imply that such approaches with mix multiple levels of abstraction can be sustained to build more complex AMR applications. Few AMR applications have been successfully build using this approach (and none with sort of flexibility that would combine AMR on grids sufficient to address complicated geometries).

Object-oriented design addresses complexity, that AMR applications are largely beyond the level of complexity that can be adequately addressed is the most significant argument for its use to simplify these sorts of applications. Within AMR++ as a library we use existing abstractions and define new one to address the complexity of AMR applications. Through the use of abstractions we address the individual requirements laid out previously.

Further motivating the introduction of abstractions we address performance of resulting applications by leveraging the semantics of these abstractions within the compilation process. Through the development of a preprocessor mechanism to introduce source code transformations we permit performance of many lower level abstractions (e.g. array objects) to be encapsulated further.

4.2 Infrastructure and Bookkeeping

Adaptive mesh refinement (AMR) computations are complicated by their dynamic nature. In the serial environment AMR requires substantial infrastructures to support the regridding processes, inter-grid operations, and local bookkeeping of positions of grids relative to one another. In the parallel environment the dynamic behavior is more problematic because it requires dynamic distribution support and load balancing. Parallel AMR is further complicated by the substantial task parallelism, in addition to the obvious data parallelism, this task parallelism requires additional infrastructure to support efficiently .

4.3 Mixed Data Parallelism and Task Parallelism

The degree of parallelism is typically dependent upon the algorithms in use and the equations being solved. Different algorithms have significant compromises between computation and communication. Substantial research work is often required to define efficient methods and suitable infrastructure. The purpose of this paper is to present AMR++ as an object-oriented library which forms a part of the OVERTURE framework, a much larger object-oriented numerical framework developed and supported at Lawrence Livermore National Laboratory and distributed on the Web for the last six years.

4.4 Geometry Issues

To be particularly useful in large-scale simulations the additional complexity of AMR and complex geometry must be addressed. To date, nearly all AMR work has addressed only single rectangular domains, though in a few cases using internal boundary mechanism to provide access to non-trivial geometries . The complexity of AMR is substantial, and has not previously permitted the additional complexity of parallelism. For example, the inclusion of AMR within the solution process on non-trivial curvilinear coordinate grids forces the refinement process to include numerous grid generation issues. Invariablely, such geometric information, the mapping from which the original curvilinear coordinate grid was built, is unavailable in the solution process which otherwise would not require it. For this reason, most AMR work is done exclusively on rectangular Cartesian grids. The addition of refinement to a curvilinear coordinate grid is not a trivial process, since it involves grid generation issues similar to construction of the grid being refined. Additional points can not be center clumsily between points on the existing curvilinear coordinate grid. When a curvilinear coordinate grid is refined, the mapping from which the curvilinear coordinate grid was built must be evaluated, so the mapping information from which the original problem was defined must be carried through the solution process so as to be available during the adaptive mesh refinement process.

Overture grid abstractions maintain their references to the grids corresponding mapping thus these abstractions are well suited for adaptive mesh refinement since the grid generation associated with the refinement of the curvilinear coordinate grids has complete information. Overture grids internally encapsulate the details of the grid generation step and provide an interface for the manual addition of local refinement independent of the grid geometry (mapping).

4.5 Data Distribution and Load Balancing

We use the Multi-Level Load Balancing algorithm to dynamically determine the distribution of different data associated with the adaptive mesh refinement grid. The distribution of data is a relegated to the P++ parallel array class library and its distributions. The support for those distributions within P++ are isolated within the PADRE library which represents an umbrella supporting a common interface for several publicly available data distribution libraries: Multi-Block PARTI, LparX, GlobalArrays, and more.

corresponding distributed array objects

5 Additional Features of AMR++

List of features in AMR++:

- High level features:
  - Parallelism (through P++)
    - Data Distributions (PADRE)
    - Load Balancing (MLB reference)
  - Complex Geometry (through Overture)
    - Simple rectangular grids
    - Curvilinear coordinate grids

        * Overlapping Grids
- Mid Level features:
  - Multiple Interfaces
  - Object-Oriented Design (Collection of Major Abstractions)
    * Solver Abstractions
    * Regridding
    * Level Transfer (and explain it)
    * Parent-Child-Sibling relationships
  - STL
  - Leverages LBL Boxlib
- Lower Level features:
  - Vertex and Cell Centering
  - Multidimensional Support
  - Different Refinement ratios
  - arbitrary Ghost cell widths
  - variable orders of accuracy for interpolation and projection
  - Gridding efficiency
    * Many details here

5.1 Features of AMR++

- built on top of A++/P++, OVERTURE, boxlib
- supports cell, vertex centered curvilinear grids
- one, two and three dimensional problems
- support for different refinement ratios
- support to write adaptive solvers
- different orders of accuracy can be specified
- ghost cell widths
- optimization via ROSE
- designed to support task and data parallelism
- current support for data parallelism
- future support for task parallelism via multi-threading
- dynamic load balancing
- seeks to maximize flexibility - 2 interfaces exist, possible for user to define other interfaces to AMR++
- powerful abstractions enable quick coding of AMR++ applications
- usual case is "complex geometries, simple physics or else complex physics, simple geometries". AMR++ seeks to provide support to solve complex problems on complex geometries by use of orthogonal abstractions
- template specialization
- code reuse

6 Application Example

Example of AMR on an overlapping grid:

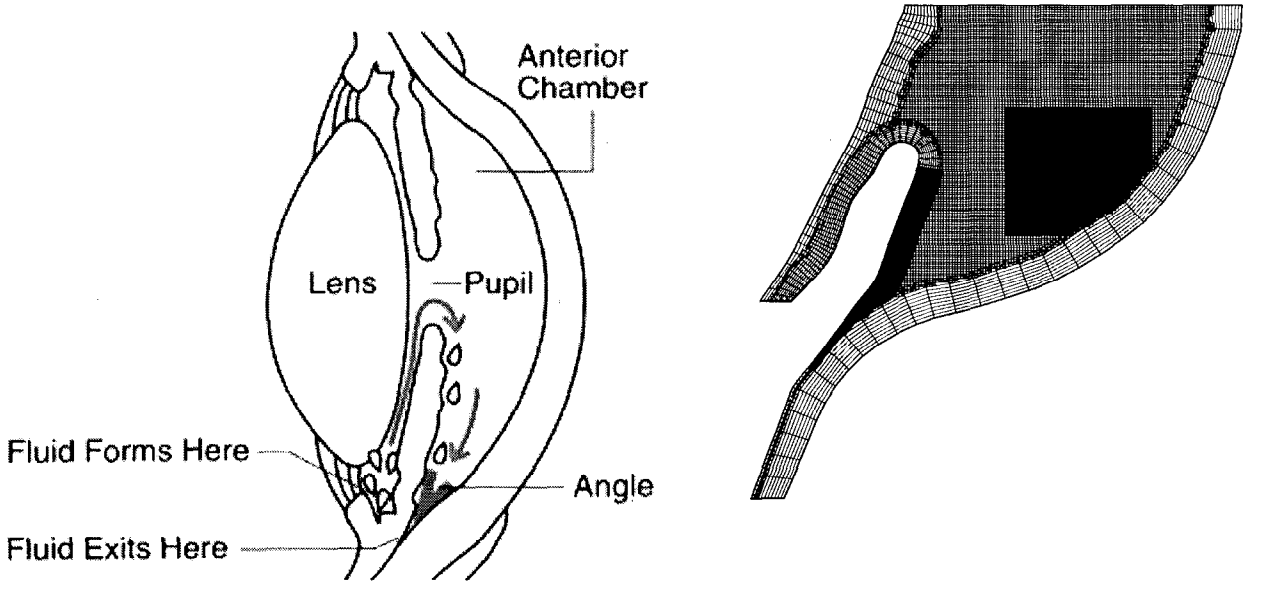

Figure 1: Eye stuff.

7 Performance of Object-Oriented Abstractions

8 Approach to Performance

The execution of array statements involves inefficiencies stemming from several sources and the problem has been well documented, by many researchers Our approach to performance within **Overture** is to use a preprocessor to introduce optimizing source-to-source transformations. The C++ source-to-source preprocessor is built using ROSE; a tool we have designed and implemented to build application specific preprocessors.

ROSE is a programmable source-to-source transformation tool built on top of SAGE for the optimization of C++ object-oriented frameworks. While we have specific goals for this work within **Overture** , ROSE applies equally well to any other object-oriented framework.

A common problem within object-oriented C++ scientific computing is that the high level semantics of abstractions introduced (e.g. parallel array objects) are ignored by the C++ compiler. Classes and overloaded operators are seen as unoptimizable structures and function calls. Such abstractions can provide for particularly simple development of large scale parallel scientific software, but the lack of optimization greatly affects the performance and utility. Because C++ lacks a mechanism to interact with the compiler, elaborate mechanisms are often implemented within such parallel frameworks to introduce complex template-based and/or runtime optimizations (such as runtime dependence analysis, deferred evaluation, runtime code generation, etc.). These approaches are however not satisfactory since they are often marginally effective, require long compile times, and/or are not sufficiently robust.

Preprocessors built using ROSE have a few features that stand out:

1. A hierarchy of grammars are specified as input to ROSE to build (tailor) the preprocessor specific to a given object-oriented application, library, or framework. ROSETTA, a code generator we have designed and implemented, is used to generate an implementation of the grammars that are used internally. The hierarchy of grammars (and their implementations) are used to construct separate program trees internally, one program tree per grammar, each representing the user's application. The program trees are edited as required to replace selected subtrees with other subtrees representing a specific transformation. Quite complex criteria may be used to identify

where transformations may be applied, this mechanism is superior to pattern-recognition of static subtrees within the program tree because it is more general and readily tailored.

2. Transformations are specified which are then built into the user application automatically where appropriate. The mechanism is designed to permit the automated introduction of particularly complex transformations (such as the cache based transformations specified in , space does not permit an elaboration of this.
3. To simplify the debugging, preprocessor's output (C++ code) is formatted identical to the input application code (except for transformations that are introduced, which have a default formatting). Numerous options are included to tailor the formatting of the output code and to simplify working with either its view directly within the debugger or its reference to the original application source within the debugger. Comments and all C preprocessor (cpp) control structures are preserved within the output C++ code.
4. The design of ROSE is simplified by leveraging both SAGE 2 and the EDG C++ front-end. EDG supplies numerous vendors with the C++ front-end for their compiler and represents the current best implementation of C++. In principle this permits the preprocessors built by ROSE to address the complete C++ language (as implemented by the best available front-end). Modifications have been made to SAGE 2 to permit portability and allow us to fulfill on a complete representation of the language. By design, we leverage many low-level optimizations provided within modern compilers while focusing on higher level optimizations largely out of reach because traditional approaches can not leverage the semantics of high level abstractions. In doing so, we slightly blur the distinction between a library or framework, a language, and a compiler. But, because we leverage several good quality tools the implementation is greatly simplified.

The approach is different from other open C++ compiler approaches because it provides a mechanism for defining high level grammars specific to an object-oriented framework and a relatively simple approach to the specification of large and complex transformations. A requirement for representing the program tree within different user defined grammars is to have access to the full program tree, this is not possible (as we best understand) within the OpenC++ research work. By using SAGE 2 and ROSE the entire program tree, represented in each grammar, is made available; this permits more sophisticated program analysis (when combined with the greater semantic knowledge of object-oriented abstractions) and more complex transformations. We believe that the techniques we have developed greatly complement the approaches represented within OpenC++, in particular the Meta object mechanism represented within that work. That SAGE is in many ways similar to the MPC++ work, we believe we could have alternatively built off of that tool in place of SAGE (though this is not clear). However, since SAGE 2 uses the EDG front-end we expect this will simplify access to the complete C++ language. MPC++ addresses more of the issues associated with easily introducing some transformations than SAGE, but not of the complexity that we require for cache based transformations . Each represent only a single grammar (the C++ grammar) and this is far too complex (we believe) a starting point for the identification of where sophisticated transformations can be introduced. The overall compile-time optimization goals are related to ideas put forward by Ian Angus , but with numerous distinguishing points:

1. We have decoupled the optimization from the back-end compiler to simplify the design.
2. We have developed hierarchies of grammars to permit arbitrarily high level abstractions to be represented with the greatest simplicity with the program tree. The use of multiple program trees (one for each grammar) serves to organize high level transformations.
3. We provide a simple mechanism to implement transformations.
4. We leverage the semantics of the abstractions to drive optimizations.
5. We have implemented and demonstrated the preprocessor approach on several large numerical applications.

Finally, because ROSE is based ultimately (through SAGE) upon the EDG C++ front-end, the full language is made available; consistent with the best of the commercial vendor C++ compilers which

most often use the same EDG C++ front-end internally. However, some aspects of the complete support of C++ within SAGE are incomplete.

9 Conclusions

The existence of parallel applications using adaptive mesh refinement in complex geometries is notably none-existent. The combination of the multiple complexities represented by AMR, geometry, and parallelism have historically proven beyond the limits of previous efforts, since no such work has been done previously. The development of AMR represents a significant complexity by itself, even for serial computing and simple Cartesian geometries. The development of the array class abstraction to permit serial code to be developed trivially and recompiled to run in parallel represents another significant complexity. Further the development of applications in complex geometries represents a third complexity. The abstractions developed within Overture provide support for geometry and specifically the development of applications on overlapping grids.

AMR++ combines these three levels of complexity and forms a part of the Overture framework.

tools to abstract the much of the complexity of applications on overlapping grids

Our AMR++ work has been done to permit our numerical research on adaptive mesh refinement algorithms. Since many meaningful applications have complex geometries and these represent our application domain AMR++ addresses the requirements of complex geometries. Importantly the development of AMR++ is made independent of geometric considerations because of the use of abstractions from the rest of Overture which encapsulate these details. Since we work on large scale applications with intense computational requirements we have had to address how to develop parallel applications. These two significant complexities greatly complicate the development of AMR. That the combination of these three complexities

Thus independent of our applications, the development of AMR++ itself demonstrates the use of multiple powerful abstractions.

10 Software availability

The Overture software is publicly available and can be obtained from the Overture home page, `http://www.llnl.gov/ca` The reports referenced herein are also available at this site.

References

[1] Quinlan, D., Berndt, M. *MLB: Multilevel Load Balancing for Structured Grid Applications*, Published in Preceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997

[2] Brown, D., Chesshire, G., Henshaw, W., and Quinlan, D., *OVERTURE: An Object-Oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments*, Published in Preceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997

[3] Balsára, D., Quinlan, D., *Parallel Object-Oriented Adaptive Mesh Refinement*, Published in Preceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997

[4] K. D. Brislawn, D. L. Brown, G. Chesshire, and J. S. Saltzman, *Adaptive composite overlapping grids for hyperbolic conservation laws*, LANL unclassified report 95-257, Los Alamos National Laboratory, 1995.

[5] M. Lemke, D. Quinlan, *P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications*, CONPAR/VAPP V, September 1992, Lyon, France; published in *Lecture Notes in Computer Science*, Springer Verlag, September 1992.

[6] R. Parsons, D. Quinlan, *Run-time Recognition of Task Parallelism within the P++ Parallel Array Class Library*, Proceedings of the Conference on Parallel Scalable Libraries, Mississippi State, 1993.